7 Rendering HTML using Razor views

This chapter covers

- Creating Razor views to display HTML to a user
- Using C# and the Razor markup syntax to generate HTML dynamically
- Reusing common code with layouts and partial views

It's easy to get confused between the terms involved in Razor Pages—`PageModel`, page handlers, Razor views—especially as some of the terms describe concrete features, and others describe patterns and concepts. We've touched on all these terms in detail in previous chapters, but it's important to get them straight in your mind:

- *Razor Pages*—Razor Pages generally refers to the page-based paradigm that combines routing, model binding, and HTML generation using Razor views.
- *Razor Page*—A single Razor Page represents a single page or "endpoint." It typically consists of two files: a .cshtml file containing the Razor view, and a .cshtml.cs file containing the page's `PageModel`.
- `PageModel`—The `PageModel` for a Razor Page is where most of the action happens. It's where you define the binding models for a page, which extracts data from the incoming request. It's also where you define the page's page handlers.

- *Page handler*—Each Razor Page typically handles a single *route*, but it can handle multiple HTTP verbs like `GET` and `POST`. Each page handler typically handles a single HTTP verb.
- *Razor view*—Razor views (also called Razor templates) are used to generate HTML. They are typically used in the final stage of a Razor Page to generate the HTML response to send back to the user.

In the previous four chapters, I've covered a whole cross section of Razor Pages, including the MVC design pattern, the Razor Page `PageModel`, page handlers, routing, and binding models. This chapter covers the last part of the MVC pattern—using a view to generate the HTML that's delivered to the user's browser.

In ASP.NET Core, views are normally created using the *Razor* markup syntax (sometimes described as a templating language), which uses a mixture of HTML and C# to generate the final HTML. This chapter covers some of the features of Razor and how to use it to build the view templates for your application. Generally speaking, users will have two sorts of interactions with your app: they'll read data that your app displays, and they'll send data or commands back to it. The Razor language contains a number of constructs that make it simple to build both types of applications.

When displaying data, you can use the Razor language to easily combine static HTML with values from your `PageModel`. Razor can use C# as a control mechanism, so adding conditional elements and loops is simple—something you couldn't achieve with HTML alone.

The normal approach to sending data to web applications is with HTML forms. Virtually every dynamic app you build will use forms; some applications will be pretty much nothing *but* forms! ASP.NET Core and the Razor templating language include a number of *Tag Helpers* that make generating HTML forms easy.

NOTE You'll get a brief glimpse of Tag Helpers in the next section, but I'll explore them in detail in the next chapter.

In this chapter we'll be focusing primarily on displaying data and generating HTML using Razor, rather than creating forms. You'll see how to render values from your `PageModel` to the HTML, and how to use C# to control the generated output. Finally, you'll learn how to extract the common elements of your views into sub-views called *layouts* and *partial views*, and how to compose them to create the final HTML page.

7.1 Views: Rendering the user interface

In this section, I provide a quick introduction to rendering HTML using Razor views. We'll recap the MVC design pattern used by Razor Pages, and where the view fits in. I'll then introduce you to how Razor syntax allows you to mix C# and HTML to generate dynamic UIs.

As you know from earlier chapters on the MVC design pattern, it's the job of the Razor Page's page handler to choose what to return to the client. For example, if you're developing a to-do list application, imagine a request to view a particular to-do item, as shown in figure 7.1.

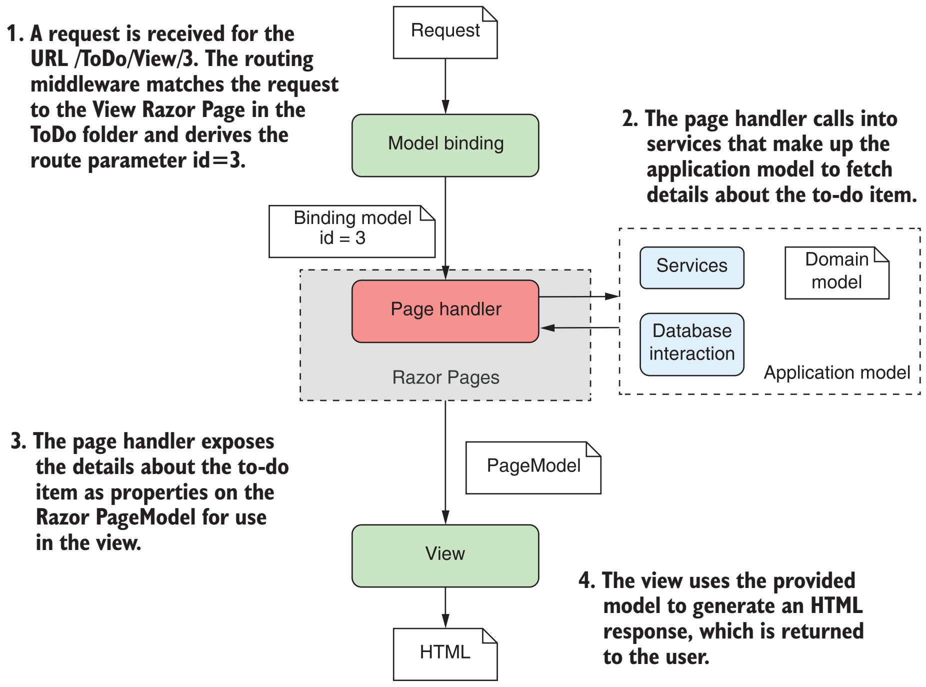

1. A request is received for the URL /ToDo/View/3. The routing middleware matches the request to the View Razor Page in the ToDo folder and derives the route parameter id=3.
Request
Model binding
Binding model id = 3
2. The page handler calls into services that make up the application model to fetch details about the to-do item.
Services
Domain model
Database interaction
Application model
Page handler
Razor Pages
3. The page handler exposes the details about the to-do item as properties on the Razor PageModel for use in the view.
PageModel
View
HTML
4. The view uses the provided model to generate an HTML response, which is returned to the user.

Figure 7.1 Handling a request for a to-do list item using ASP.NET Core Razor Pages. The page handler builds the data required by the view and exposes it as properties on the `PageModel`. The view generates HTML based only on the data provided; it doesn't need to know where that data come from.

A typical request follows the steps shown in figure 7.1:

1. The middleware pipeline receives the request, and the routing middleware determines the endpoint to invoke—in this case, the `View` Razor Page in the ToDo folder.
2. The model binder (part of the Razor Pages framework) uses the request to build the binding models for the page, as you saw in the previous chapter. The binding models are set as properties on the Razor Page or are passed to the page handler method as arguments when the handler is executed. The page handler checks that you have passed a valid `id` for the to-do item, making the request valid.
3. Assuming all looks good, the page handler calls out to the various services that make up the application model. This might load the details about the to-do

from a database, or from the filesystem, returning them to the handler. As part of this process, either the application model or the page handler itself generates values to pass to the view and sets them as properties on the Razor Page `PageModel`.

Once the page handler has executed, the `PageModel` should contain all the data required to render a view. In this example, it contains details about the to-do itself, but it might also contain other data: how many to-dos you have left, whether you have any to-dos scheduled for today, your username, and so on—anything that controls how to generate the end UI for the request.

4 The Razor view template uses the `PageModel` to generate the final response and returns it to the user via the middleware pipeline.

A common thread throughout this discussion of MVC is the separation of concerns MVC brings, and it's no different when it comes to your views. It would be easy enough to directly generate the HTML in your application model or in your controller actions, but instead you delegate that responsibility to a single component, the view.

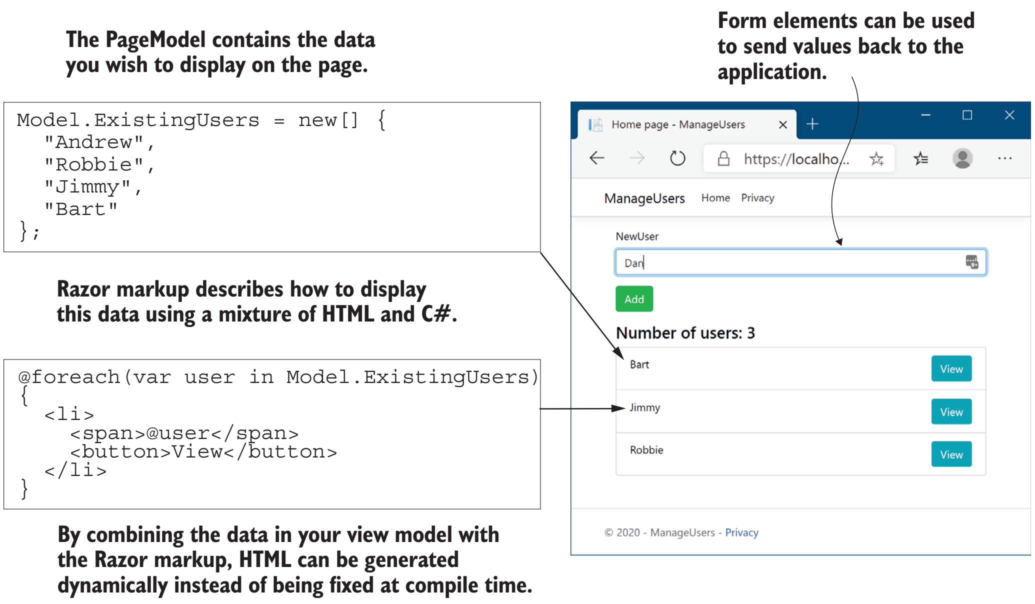
But even more than that, you'll also separate the *data* required to build the view from the *process* of building it by using properties on the `PageModel`. These properties should contain all the dynamic data needed by the view to generate the final output.

TIP Views shouldn't call methods on the `PageModel`—the view should generally only be accessing data that has already been collected and exposed as properties.

Razor Page handlers indicate that the Razor view should be rendered by returning a `PageResult` (or by returning `void`), as you saw in chapter 4. The Razor Pages infrastructure executes the Razor view associated with a given Razor Page to generate the final response. The use of C# in the Razor template means you can dynamically generate the final HTML sent to the browser. This allows you to, for example, display the name of the current user in the page, hide links the current user doesn't have access to, or render a button for every item in a list.

Imagine your boss asks you to add a page to your application that displays a list of the application's users. You should also be able to view a user from the page, or create a new one, as shown in figure 7.2.

With Razor templates, generating this sort of dynamic content is simple. For example, listing 7.1 shows a template that could be used to generate the interface in figure 7.2. It combines standard HTML with C# statements and uses Tag Helpers to generate the form elements.



The PageModel contains the data you wish to display on the page.

```
Model.ExistingUsers = new[] {
  "Andrew",
  "Robbie",
  "Jimmy",
  "Bart"
};
```

Razor markup describes how to display this data using a mixture of HTML and C#.

```
@foreach(var user in Model.ExistingUsers)
{
  <li>
    <span>@user</span>
    <button>View</button>
  </li>
}
```

By combining the data in your view model with the Razor markup, HTML can be generated dynamically instead of being fixed at compile time.

Form elements can be used to send values back to the application.

Figure 7.2 The use of C# in Razor lets you easily generate dynamic HTML that varies at runtime. In this example, using a `foreach` loop inside the Razor view dramatically reduces the duplication in the HTML that you would otherwise have to write.

Listing 7.1 A Razor template to list users, and a form for adding a new user

```
@page
@model IndexViewModel
<div class="row">
<div class="col-md-6">
<form method="post">
    <div class="form-group">
        <label asp-for="NewUser"></label>
        <input class="form-control" asp-for="NewUser" />
        <span asp-validation-for="NewUser"></span>
    </div>
    <div class="form-group">
        <button type="submit"
          class="btn btn-success">Add</button>
    </div>
</form>
</div>
</div>

<h4>Number of users: @Model.ExistingUsers.Count</h4>
<div class="row">
<div class="col-md-6">
<ul class="list-group">
@foreach (var user in Model.ExistingUsers)
```

Normal HTML is sent to the browser unchanged.

Tag Helpers attach to HTML elements to create forms.

Values can be written from C# objects to the HTML.

C# constructs like for loops can be used in Razor.

```
{
<li class="list-group-item d-flex justify-content-between">
    <span>@user</span>
    <a class="btn btn-info"
        asp-page="ViewUser"
        asp-route-userName="@user">View</a>
</li>
}
</ul>
</div>
</div>
```

Tag Helpers can also be used outside of forms to help in other HTML generation.

This example demonstrates a variety of Razor features. There's a mixture of HTML that's written unmodified to the response output, and there are various C# constructs used to dynamically generate HTML. In addition, you can see several Tag Helpers. These look like normal HTML attributes that start `asp-`, but they're part of the Razor language. They can customize the HTML element they're attached to, changing how it's rendered. They make building HTML forms much simpler than they would be otherwise. Don't worry if this template is a bit overwhelming at the moment; we'll break it all down as you progress through this chapter and the next.

Razor Pages are compiled when you build your application. Behind the scenes, they become just another C# class in your application. It's also possible to enable *runtime* compilation of your Razor Pages. This allows you to modify your Razor Pages while your app is running, without having to explicitly stop and rebuild. This can be handy when developing locally, but it's best avoided when you deploy to production.[1]

NOTE Like most things in ASP.NET Core, it's possible to swap out the Razor templating engine and replace it with your own server-side rendering engine. You can't replace Razor with a client-side framework like Angular or React. If you want to take this approach, you'd use Web APIs instead. I'll discuss Web APIs in detail in chapter 9.

In the next section we'll look in more detail at how Razor views fit into the Razor Pages framework, and how you can pass data from your Razor Page handlers to the Razor view to help build the HTML response.

7.2 Creating Razor views

In this section we'll look at how Razor views fit into the Razor Pages framework. You'll learn how to pass data from your page handlers to your Razor views, and how you can use that data to generate dynamic HTML.

With ASP.NET Core, whenever you need to display an HTML response to the user, you should use a view to generate it. Although it's possible to directly generate a `string` from your page handlers, which will be rendered as HTML in the browser, this

[1] For details on enabling runtime compilation, including enabling conditional precompilation for production environments, see the documentation: http://mng.bz/opwy.

approach doesn't adhere to the MVC separation of concerns and will quickly leave you tearing your hair out.

NOTE Some middleware, such as the `WelcomePageMiddleware` you saw in chapter 3, may generate HTML responses without using a view, which can make sense in some situations. But your Razor Page and MVC controllers should always generate HTML using views.

Instead, by relying on Razor views to generate the response, you get access to a wide variety of features, as well as editor tooling to help. This section serves as a gentle introduction to Razor views, the things you can do with them, and the various ways you can pass data to them.

7.2.1 Razor views and code-behind

In this book you've already seen that Razor Pages typically consist of two files:

- The .cshtml file, commonly called the *Razor view*.
- The .cshtml.cs file, commonly called the *code-behind*, which contains the `PageModel`.

The Razor view contains the `@page` *directive*, which *makes it* a Razor Page, as you saw in chapter 4. Without this directive, the Razor Pages framework will not route requests to the page, and the file will be ignored for most purposes.

DEFINITION A *directive* is a statement in a Razor file that changes the way the template is parsed or compiled. Another common directive is the `@using newNamespace` directive, which makes objects in the `newNamespace` namespace available.

The code-behind .cshtml.cs file contains the `PageModel` for an associated Razor Page. It contains the page handlers that respond to requests, and it is where the Razor Page typically interacts with other parts of your application.

Even though the .cshtml and .cshtml.cs files share the same name, such as ToDoItem.cshtml and ToDoItem.cshtml.cs, it's not the filename that's linking them together. But if it's not by filename, how does the Razor Pages framework know which `PageModel` is associated with a given Razor Page view file?

At the top of each Razor Page, just after the `@page` directive, is an `@model` directive with a `Type`, indicating which `PageModel` is associated with the Razor view. For example, the following directives indicate that the `ToDoItemModel` is the `PageModel` associated with the Razor Page:

```
@page
@model ToDoItemModel
```

Once a request is routed to a Razor Page, as we covered in chapter 5, the framework looks for the `@model` directive to decide which `PageModel` to use. Based on the `PageModel` selected, it then binds to any properties in the `PageModel` marked with the

[BindProperty] attribute (as we covered in chapter 6) and executes the appropriate page handler (based on the request's HTTP verb).

NOTE Technically, the PageModel and @model directive are optional. If you don't specify a PageModel, the framework will execute a default page handler, as you saw in chapter 5. It's also possible to combine the .cshtml and .cshtml.cs files into a single .cshtml file. In practice, neither of these approaches are very common, even for simple pages, but it's something to be aware of if you run into it.[2]

In addition to the @page and @model directives, the Razor view file contains the Razor template that is executed to generate the HTML response.

7.2.2 Introducing Razor templates

Razor view templates contain a mixture of HTML and C# code interspersed with one another. The HTML markup lets you easily describe exactly what should be sent to the browser, whereas the C# code can be used to dynamically change what is rendered. For example, the following listing shows an example of Razor rendering a list of strings, representing to-do items.

Listing 7.2 Razor template for rendering a list of strings

```
@page
@{
    var tasks = new List<string>
      { "Buy milk", "Buy eggs", "Buy bread" };
}
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i< tasks.Count; i++)
{
  var task = tasks[i];
  <li>@i - @task</li>
}
</ul>
```

Arbitrary C# can be executed in a template. Variables remain in scope throughout the page.

Standard HTML markup will be rendered to the output unchanged.

Mixing C# and HTML allows you to dynamically create HTML at runtime.

The pure HTML sections in this template are in the angle brackets. The Razor engine copies this HTML directly to the output, unchanged, as though you were writing a normal HTML file.

NOTE The ability of Razor syntax to know when you are switching between HTML and C# can be both uncanny and infuriating at times. I discuss the details on how to control this transition in section 7.3.

As well as HTML, you can also see a number of C# statements in there. The advantage of being able to, for example, use a for loop rather than having to explicitly write out

[2] These alternative approaches are not generally considered idiomatic, so I don't discuss them in this book, but you can read more about them here: https://www.learnrazorpages.com/razor-pages.

each `<li>` element should be self-evident. I'll dive a little deeper into more of the C# features of Razor in the next section. When rendered, the template in listing 7.2 would produce the following HTML.

Listing 7.3 HTML output produced by rendering a Razor template

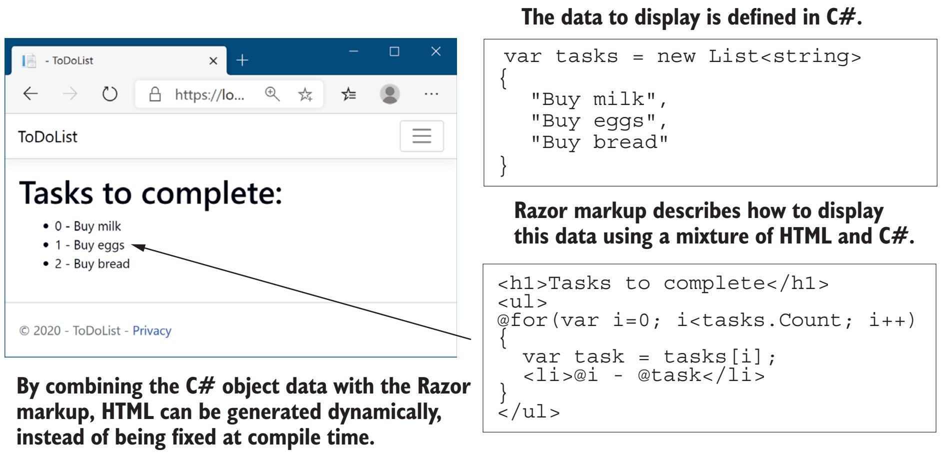
```
<h1>Tasks to complete</h1>
<ul>
  <li>0 - Buy milk</li>
  <li>1 - Buy eggs</li>
  <li>2 - Buy bread</li>
</ul>
```

HTML from the Razor template is written directly to the output.

The `<li>` elements are generated dynamically based on the data.

HTML from the Razor template is written directly to the output.

As you can see, the final output of a Razor template after it's been rendered is simple HTML. There's nothing complicated left, just straight HTML markup that can be sent to the browser and rendered. Figure 7.3 shows how a browser would render it.


- ToDoList
https://lo...
ToDoList
Tasks to complete:
0 - Buy milk
1 - Buy eggs
2 - Buy bread
© 2020 - ToDoList - Privacy
The data to display is defined in C#.
Razor markup describes how to display this data using a mixture of HTML and C#.
By combining the C# object data with the Razor markup, HTML can be generated dynamically, instead of being fixed at compile time.

```
var tasks = new List<string>
{
   "Buy milk",
   "Buy eggs",
   "Buy bread"
}
```

```
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i<tasks.Count; i++)
{
  var task = tasks[i];
  <li>@i - @task</li>
}
</ul>
```

Figure 7.3 Razor templates can be used to generate the HTML dynamically at runtime from C# objects. In this case, a `for` loop is used to create repetitive HTML `<li>` elements.

In this example, I hardcoded the list values for simplicity—there was no dynamic data provided. This is often the case on simple Razor Pages, like what you might have on your home page—you need to display an almost static page. For the rest of your application, it will be far more common to have some sort of data you need to display, typically exposed as properties on your `PageModel`.

7.2.3 Passing data to views

In ASP.NET Core, you have several ways of passing data from a page handler in a Razor Page to its view. Which approach is best will depend on the data you're trying to pass through, but in general you should use the mechanisms in the following order:

- `PageModel` *properties*—You should generally expose any data that needs to be displayed as properties on your `PageModel`. Any data that is specific to the associated Razor view should be exposed this way. The `PageModel` object is available in the view when it's rendered, as you'll see shortly.
- `ViewData`—This is a dictionary of objects with `string` keys that can be used to pass arbitrary data from the page handler to the view. In addition, it allows you to pass data to *_layout* files, as you'll see in section 7.4. This is the main reason for using `ViewData` instead of setting properties on the `PageModel`.
- `HttpContext`—Technically, the `HttpContext` object is available in both the page handler and Razor view, so you *could* use it to transfer data between them. But don't—there's no need for it with the other methods available to you.
- `@inject` *services*—You can use dependency injection to make services available in your views, though this should normally be used very sparingly. I describe dependency injection and the `@inject` directive in chapter 10.

Far and away the best approach for passing data from a page handler to a view is to use properties on the `PageModel`. There's nothing special about the properties themselves; you can store anything there to hold the data you require.

> **NOTE** Many frameworks have the concept of a data context for binding UI components. The `PageModel` is a similar concept, in that it contains values to display in the UI, but the binding is only one-directional; the `PageModel` provides values to the UI, and once the UI is built and sent as a response, the `PageModel` is destroyed.

As I described in section 7.2.1, the `@model` directive at the top of your Razor view describes which `Type` of `PageModel` is associated with a given Razor Page. The `PageModel` associated with a Razor Page contains one or more page handlers, and exposes data as properties for use in the Razor view.

Listing 7.4 Exposing data as properties on a `PageModel`

```
public class ToDoItemModel : PageModel
{
    public List<string> Tasks { get; set; }
    public string Title { get; set; }

    public void OnGet(int id)
    {
```

The PageModel is passed to the Razor view when it executes.

The public properties can be accessed from the Razor view.

```
        Title = "Tasks for today";
        Tasks = new List<string>              Building the required
        {                                     data: this would
            "Get fuel",                       normally call out to a
            "Check oil",                      service or database
            "Check tyre pressure"             to load the data.
        };
    }
}
```

You can access the `PageModel` instance itself from the Razor view using the `Model` property. For example, to display the `Title` property of the `ToDoItemModel` in the Razor view, you'd use `<h1>@Model.Title</h1>`. This would render the string provided in the `ToDoItemModel.Title` property, producing the `<h1>Tasks for today</h1>` HTML.

TIP Note that the `@model` directive should be at the top of your view, just after the `@page` directive, and it has a lowercase `m`. The `Model` property can be accessed anywhere in the view and has an uppercase `M`.

In the vast majority of cases, using public properties on your `PageModel` is the way to go; it's the standard mechanism for passing data between the page handler and the view. But in some circumstances, properties on your `PageModel` might not be the best fit. This is often the case when you want to pass data between view layouts (you'll see how this works in section 7.4).

A common example is the title of the page. You need to provide a title for every page in your application, so you *could* create a base class with a `Title` property and make every `PageModel` inherit from it. But that's very cumbersome, so a common approach for this situation is to use the `ViewData` collection to pass data around.

In fact, the standard Razor Page templates use this approach by default by setting values on the `ViewData` dictionary from within the view itself:

```
@{
    ViewData["Title"] = "Home Page";
}
<h2>@ViewData["Title"].</h2>
```

This template sets the value of the `"Title"` key in the `ViewData` dictionary to `"Home Page"` and then fetches the key to render in the template. This set and immediate fetch might seem superfluous, but as the `ViewData` dictionary is shared throughout the request, it makes the title of the page available in layouts, as you'll see later. When rendered, the preceding template would produce the following output:

```
<h2>Home Page.</h2>
```

You can also set values in the `ViewData` dictionary from your page handlers in two different ways, as shown in the following listing.

Listing 7.5 Setting `ViewData` values using an attribute

```
public class IndexModel: PageModel
{
    [ViewData]
    public string Title { get; set; }

    public void OnGet()
    {
        Title = "Home Page";
        ViewData["Subtitle"] = "Welcome";
    }
}
```

Properties marked with the [ViewData] attribute are set in the ViewData.

The value of ViewData["Title"] will be set to "Home Page".

You can set keys in the ViewData dictionary directly.

You can display the values in the template in the same way as before:

```
<h1>@ViewData["Title"]</h3>
<h2>@ViewData["Subtitle"]</h3>
```

TIP I don't find the `[ViewData]` attribute especially useful, but it's another feature to look out for. Instead, I create a set of global, static constants for any `ViewData` keys, and I reference those instead of typing `"Title"` repeatedly. You'll get IntelliSense for the values, they'll be refactor-safe, and you'll avoid hard-to-spot typos.

As I mentioned previously, there are other mechanisms besides `PageModel` properties and `ViewData` that you can use to pass data around, but these two are the only ones I use personally, as you can do everything you need with them. As a reminder, always use `PageModel` properties where possible, as you benefit from strong typing and IntelliSense. Only fall back to `ViewData` for values that need to be accessed *outside* of your Razor view.

You've had a small taste of the power available to you in Razor templates, but in the next section we'll dive a little deeper into some of the available C# capabilities.

7.3 Creating dynamic web pages with Razor

You might be glad to know that pretty much anything you can do in C# is possible in Razor syntax. Under the covers, the .cshtml files are compiled into normal C# code (with `string` for the raw HTML sections), so whatever weird and wonderful behavior you need can be created!

Having said that, just because you *can* do something doesn't mean you *should*. You'll find it much easier to work with, and maintain, your files if you keep them as simple as possible. This is true of pretty much all programming, but I find it to be especially so with Razor templates.

This section covers some of the more common C# constructs you can use. If you find you need to achieve something a bit more exotic, refer to the Razor syntax documentation at http://mng.bz/opj2.

7.3.1 Using C# in Razor templates

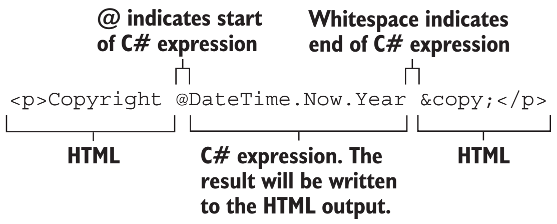
One of the most common requirements when working with Razor templates is to render a value you've calculated in C# to the HTML. For example, you might want to print the current year to use with a copyright statement in your HTML, to give this result:

```
<p>Copyright 2020 ©</p>
```

Or you might want to print the result of a calculation:

```
<p>The sum of 1 and 2 is <i>3</i><p>
```

You can do this in two ways, depending on the exact C# code you need to execute. If the code is a single statement, you can use the `@` symbol to indicate you want to write the result to the HTML output, as shown in figure 7.4. You've already seen this used to write out values from the `PageModel` or from `ViewData`.

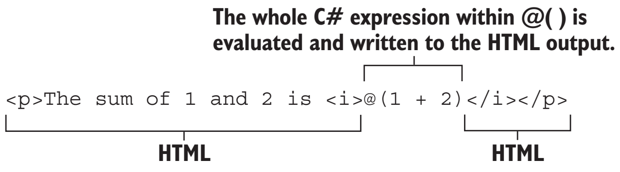


```
<p>Copyright @DateTime.Now.Year &copy;</p>
```

@ indicates start of C# expression
Whitespace indicates end of C# expression
HTML
C# expression. The result will be written to the HTML output.
HTML

Figure 7.4 Writing the result of a C# expression to HTML. The @ symbol indicates where the C# code begins, and the expression ends at the end of the statement, in this case at the space.

If the C# you want to execute is something that *needs* a space, then you need to use parentheses to demarcate the C#, as shown in figure 7.5.



```
<p>The sum of 1 and 2 is <i>@(1 + 2)</i></p>
```

The whole C# expression within @( ) is evaluated and written to the HTML output.
HTML
HTML

Figure 7.5 When a C# expression contains whitespace, you must wrap it in parentheses using `@()` so the Razor engine knows where the C# stops and HTML begins.

These two approaches, in which C# is evaluated and written directly to the HTML output, are called *Razor expressions*.

> **TIP** If you want to write a literal `@` character, rather than a C# expression, use a second `@` character: `@@`.

Sometimes you'll want to execute some C#, but you don't need to output the values. We used this technique when we were setting values in `ViewData`:

```
@{
    ViewData["Title"] = "Home Page";
}
```

This example demonstrates a *Razor code block*, which is normal C# code, identified by the `@{}` structure. Nothing is written to the HTML output here; it's all compiled as though you'd written it in any other normal C# file.

> **TIP** When you execute code within code blocks, it must be valid C#, so you need to add semicolons. Conversely, when you're writing values directly to the response using Razor expressions, you don't need them. If your output HTML breaks unexpectedly, keep an eye out for missing or rogue extra semicolons.

Razor expressions are one of the most common ways of writing data from your `PageModel` to the HTML output. You'll see the other approach, using Tag Helpers, in the next chapter. Razor's capabilities extend far further than this, however, as you'll see in the next section, where you'll learn how to include traditional C# structures in your templates.

7.3.2 Adding loops and conditionals to Razor templates

One of the biggest advantages of using Razor templates over static HTML is the ability to dynamically generate the output. Being able to write values from your `PageModel` to the HTML using Razor expressions is a key part of that, but another common use is loops and conditionals. With these, you can hide sections of the UI, or produce HTML for every item in a list, for example.

Loops and conditionals include constructs such as `if` and `for` loops. Using them in Razor templates is almost identical to C#, but you need to prefix their usage with the `@` symbol. In case you're not getting the hang of Razor yet, when in doubt, throw in another `@`!

One of the big advantages of Razor in the context of ASP.NET Core is that it uses languages you're already familiar with: C# and HTML. There's no need to learn a whole new set of primitives for some other templating language: it's the same `if`, `foreach`, and `while` constructs you already know. And when you don't need them, you're writing raw HTML, so you can see exactly what the user will be getting in their browser.

In listing 7.6, I've applied a number of these different techniques in the template for displaying a to-do item. The `PageModel` has a `bool IsComplete` property, as well as a `List<string>` property called `Tasks`, which contains any outstanding tasks.

Listing 7.6 Razor template for rendering a `ToDoItemViewModel`

```
@page
@model ToDoItemModel        <- The @model directive indicates
                               the type of PageModel in Model.
```

```
<div>
    @if (Model.IsComplete)
    {
        <strong>Well done, you're all done!</strong>
    }
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.Tasks)
            {
                <li>@task</li>
            }
        </ul>
    }
</div>
```

The if control structure checks the value of the PageModel's IsComplete property at runtime.

The foreach structure will generate the <li> elements once for each task in Model.Tasks.

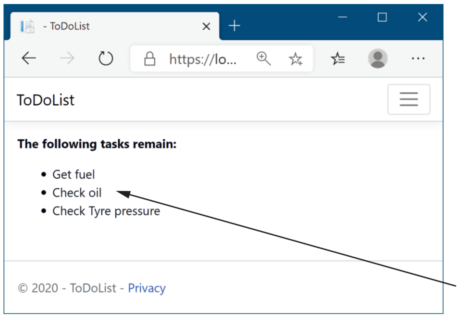
A Razor expression is used to write the task to the HTML output.

This code definitely lives up to the promise of mixing C# and HTML! There are traditional C# control structures, like `if` and `foreach`, that you'd expect in any normal program, interspersed with the HTML markup that you want to send to the browser. As you can see, the `@` symbol is used to indicate when you're starting a control statement, but you generally let the Razor template infer when you're switching back and forth between HTML and C#.

The template shows how to generate dynamic HTML at runtime, depending on the exact data provided. If the model has outstanding `Tasks`, the HTML will generate a list item for each task, producing output something like that shown in figure 7.6.

The data to display is defined on properties in the PageModel.

```
Model.IsComplete = false;
Model.Tasks = new List<string>
{
  "Get fuel",
  "Check oil",
  "Check Tyre pressure"
};
```



The following tasks remain:

- Get fuel
- Check oil
- Check Tyre pressure

© 2020 - ToDoList - Privacy

Razor markup can include C#constr ucts such as if statements and for loops.

```
@if (Model.IsComplete)
{
    <p>Well done, you're all done!</p>
} else {
    <p>The following tasks remain:</p>
    <ul>
    @foreach(var task in Model.Tasks)
    {
      <li>@task</li>
    }
    </ul>
}
```

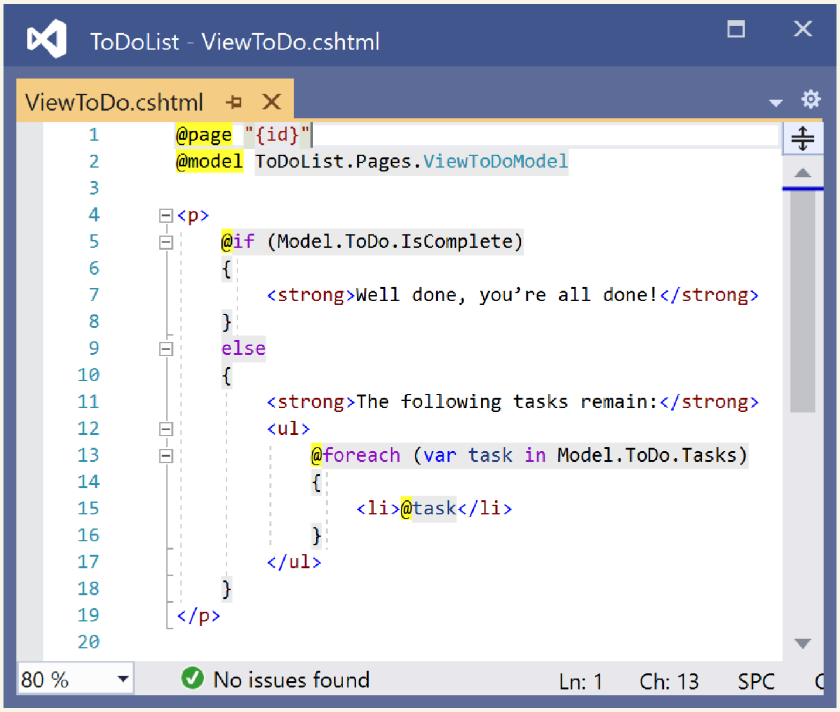
Only the relevant "if" block is rendered to the HTML, and the content within a foreach loop is rendered once for every item.

Figure 7.6 The Razor template generates a `<li>` item for each remaining task, depending on the data passed to the view at runtime. You can use an `if` block to render completely different HTML depending on the values in your model.

IntelliSense and tooling support

The mixture of C# and HTML might seem hard to read in the book, and that's a reasonable complaint. It's also another valid argument for trying to keep your Razor templates as simple as possible.

Luckily, if you're using an editor like Visual Studio or Visual Studio Code, the tooling can help somewhat. As you can see in this figure, the C# portions of the code are shaded to help distinguish them from the surrounding HTML.



```
@page "{id}"
@model ToDoList.Pages.ViewToDoModel

<p>
    @if (Model.ToDo.IsComplete)
    {
        <strong>Well done, you're all done!</strong>
    }
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.ToDo.Tasks)
            {
                <li>@task</li>
            }
        </ul>
    }
</p>
```

Visual Studio shades the C# regions of code and highlights @ symbols where C# transitions to HTML. This makes the Razor templates easier to read.

Although the ability to use loops and conditionals is powerful—they're one of the advantages of Razor over static HTML—they also add to the complexity of your view. Try to limit the amount of logic in your views to make them as easy to understand and maintain as possible.

A common trope of the ASP.NET Core team is that they try to ensure you "fall into the pit of success" when building an application. This refers to the idea that, by default, the *easiest* way to do something should be the *correct* way of doing it. This is a great philosophy, as it means you shouldn't get burned by, for example, security problems if you follow the standard approaches. Occasionally, however, you may need to step beyond the safety rails; a common use case is when you need to render some HTML contained in a C# object to the output, as you'll see in the next section.

7.3.3 Rendering HTML with Raw

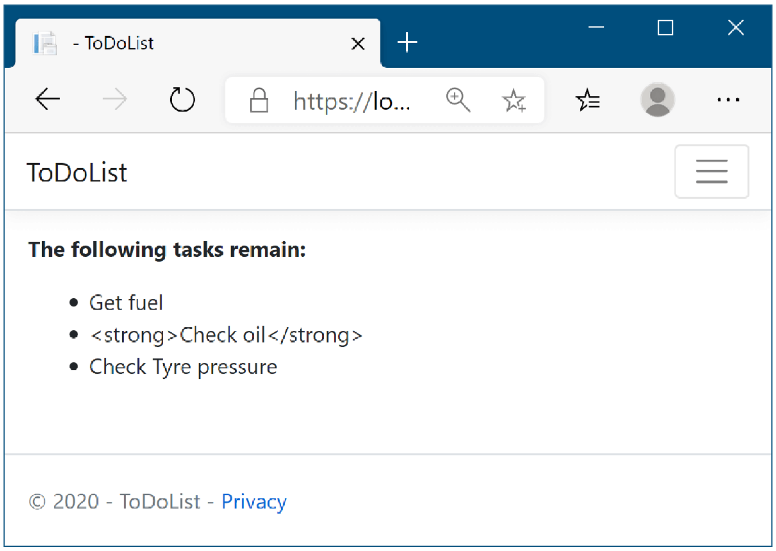
In the previous example, we rendered the list of tasks to HTML by writing the `string task` using the `@task` Razor expression. But what if the `task` variable contains HTML you want to display, so instead of `"Check oil"` it contains `"<strong>Check oil</strong>"`? If you use a Razor expression to output this as you did previously, you might hope to get this:

```
<li><strong>Check oil</strong></li>
```

But that's not the case. The HTML generated comes out like this:

```
<li>&lt;strong&gt;Check oil&lt;/strong&gt;</li>
```

Hmm, looks odd, right? What's happened here? Why did the template not write your variable to the HTML, like it has in previous examples? If you look at how a browser displays this HTML, like in figure 7.7, then hopefully it makes more sense.



The following tasks remain:

- Get fuel
- <strong>Check oil</strong>
- Check Tyre pressure

© 2020 - ToDoList - Privacy

Figure 7.7 The second item, `"<strong>Check oil<strong>"` has been HTML-encoded, so the `<strong>` elements are visible to the user as part of the task. This avoids any security issues, as users can't inject malicious scripts into your HTML.

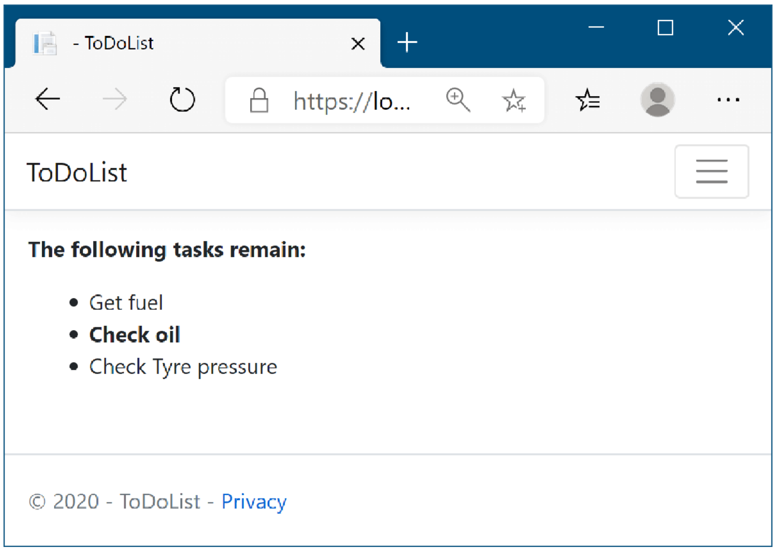
Razor templates encode C# expressions before they're written to the output stream. This is primarily for security reasons; writing out arbitrary strings to your HTML could allow users to inject malicious data and JavaScript into your website. Consequently, the C# variables you print in your Razor template get written as HTML-encoded values.

In some cases you might need to directly write out HTML contained in a `string` to the response. If you find yourself in this situation, first, stop. Do you *really* need to do this? If the values you're writing have been entered by a user, or were created based on values provided by users, there's a serious risk of creating a security hole in your website.

If you *really* need to write the variable out to the HTML stream, you can do so using the `Html` property on the view page and calling the `Raw` method:

```
<li>@Html.Raw(task)</li>
```

With this approach, the string in `task` will be directly written to the output stream, producing the HTML you originally wanted, `<li><strong>Check oil</strong></li>`, which renders as shown in figure 7.8.



The following tasks remain:

- Get fuel
- **Check oil**
- Check Tyre pressure

© 2020 - ToDoList - Privacy

Figure 7.8 The second item, "`<strong>Check oil<strong>`" has been output using `Html.Raw()`, so it hasn't been HTML-encoded. The `<strong>` elements result in the second item being shown in bold instead. Using `Html.Raw()` in this way should be avoided where possible, as it is a security risk.

WARNING Using `Html.Raw` on user input creates a security risk that users could use to inject malicious code into your website. Avoid using `Html.Raw` if possible.

The C# constructs shown in this section can be useful, but they can make your templates harder to read. It's generally easier to understand the intention of Razor templates that are predominantly HTML markup rather than C#.

In the previous version of ASP.NET, these constructs, and in particular the `Html` helper property, were the standard way to generate dynamic markup. You can still use this approach in ASP.NET Core by using the various `HtmlHelper`[3] methods on the `Html` property, but these have largely been superseded by a cleaner technique: Tag Helpers.

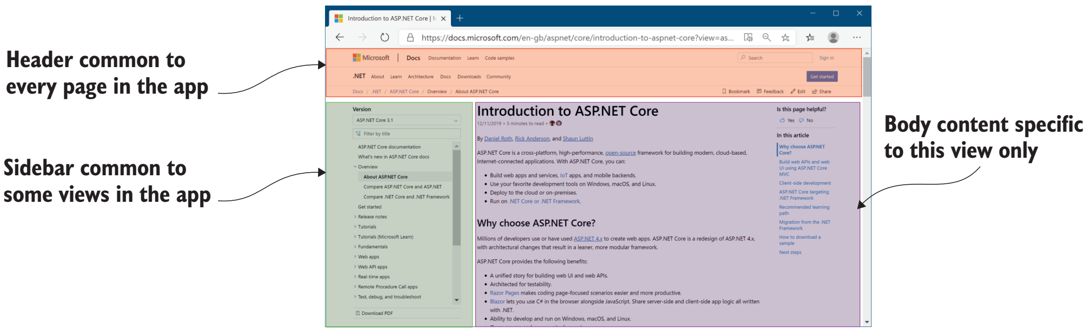
> **NOTE** I'll discuss Tag Helpers, and how to use them to build HTML forms, in the next chapter.

Tag Helpers are a useful feature that's new to Razor in ASP.NET Core, but a number of other features have been carried through from the previous version of ASP.NET. In the next section of this chapter, you'll see how you can create nested Razor templates and use partial views to reduce the amount of duplication in your views.

7.4 Layouts, partial views, and _ViewStart

In this section you'll learn about layouts and partial views, which allow you to extract common code to reduce duplication. These files make it easier to make changes to your HTML that affect multiple pages at once. You'll also learn how to run common code for every Razor Page using _ViewStart and _ViewImports, and how to include optional sections in your pages.

Every HTML document has a certain number of elements that are required: `<html>`, `<head>`, and `<body>`. As well, there are often common sections that are repeated on every page of your application, such as the header and footer, as shown in figure 7.9. Each page in your application will also probably reference the same CSS and JavaScript files.


Header common to every page in the app
Sidebar common to some views in the app
Body content specific to this view only
Introduction to ASP.NET Core | M
https://docs.microsoft.com/en-gb/aspnet/core/introduction-to-aspnet-core?view=as...
Microsoft | Docs Documentation Learn Code samples
Search
Sign in
.NET About Learn Architecture Docs Downloads Community
Get started
Docs / .NET / ASP.NET Core / Overview / About ASP.NET Core
Bookmark Feedback Edit Share
Version
ASP.NET Core 3.1
Filter by title
ASP.NET Core documentation
What's new in ASP.NET Core docs
Overview
About ASP.NET Core
Compare ASP.NET Core and ASP.NET
Compare .NET Core and .NET Framework
Get started
Release notes
Tutorials
Tutorials (Microsoft Learn)
Fundamentals
Web apps
Web API apps
Real-time apps
Remote Procedure Call apps
Test, debug, and troubleshoot
Download PDF
Is this page helpful?
Yes No
In this article
Why choose ASP.NET Core?
Build web APIs and web UI using ASP.NET Core MVC
Client-side development
ASP.NET Core targeting .NET Framework
Recommended learning path
Migration from the .NET Framework
How to download a sample
Next steps

Introduction to ASP.NET Core

12/11/2019 • 5 minutes to read •

By Daniel Roth, Rick Anderson, and Shaun Luttin

ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, Internet-connected applications. With ASP.NET Core, you can:

- Build web apps and services, IoT apps, and mobile backends.
- Use your favorite development tools on Windows, macOS, and Linux.
- Deploy to the cloud or on-premises.
- Run on .NET Core or .NET Framework.

Why choose ASP.NET Core?

Millions of developers use or have used ASP.NET 4.x to create web apps. ASP.NET Core is a redesign of ASP.NET 4.x, with architectural changes that result in a leaner, more modular framework.

ASP.NET Core provides the following benefits:

- A unified story for building web UI and web APIs.
- Architected for testability.
- Razor Pages makes coding page-focused scenarios easier and more productive.
- Blazor lets you use C# in the browser alongside JavaScript. Share server-side and client-side app logic all written with .NET.
- Ability to develop and run on Windows, macOS, and Linux.

Figure 7.9 A typical web application has a block-based layout, where some blocks are common to every page of your application. The header block will likely be identical across your whole application, but the sidebar may only be identical for the pages in one section. The body content will differ for every page in your application.

[3] `HTMLHelpers` are almost obsolete, though they're still available if you prefer to use them.

All these different elements add up to a maintenance nightmare. If you had to manually include these in every view, then making any changes would be a laborious, error-prone process involving editing every page. Instead, Razor lets you extract these common elements into *layouts*.

> **DEFINITION** A *layout* in Razor is a template that includes common code. It can't be rendered directly, but it can be rendered in conjunction with normal Razor views.

By extracting your common markup into layouts, you can reduce the duplication in your app. This makes changes easier, makes your views easier to manage and maintain, and is generally good practice!

7.4.1 Using layouts for shared markup

Layout files are, for the most part, normal Razor templates that contain markup common to more than one page. An ASP.NET Core app can have multiple layouts, and layouts can reference other layouts. A common use for this is to have different layouts for different sections of your application. For example, an e-commerce website might use a three-column view for most pages, but a single-column layout when you come to the checkout pages, as shown in figure 7.10.

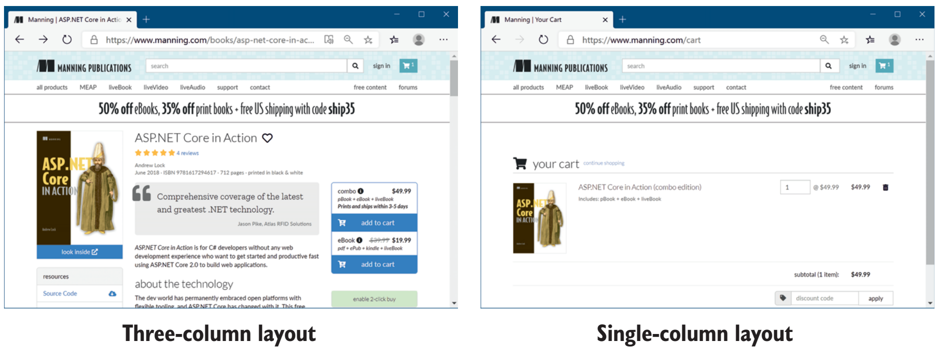


Figure 7.10 The https://manning.com website uses different layouts for different parts of the web application. The product pages use a three-column layout, but the cart page uses a single-column layout.

You'll often use layouts across many different Razor Pages, so they're typically placed in the Pages/Shared folder. You can name them anything you like, but there's a common convention to use _Layout.cshtml as the filename for the base layout in your application. This is the default name used by the Razor Page templates in Visual Studio and the .NET CLI.

TIP A common convention is to prefix your layout files with an underscore (_) to distinguish them from standard Razor templates in your Pages folder.

A layout file looks similar to a normal Razor template, with one exception: every layout must call the `@RenderBody()` function. This tells the templating engine where to insert the content from the child views. A simple layout is shown in the following listing. Typically, your application will reference all your CSS and JavaScript files in the layout, as well as include all the common elements, such as headers and footers, but this example includes pretty much the bare minimum HTML.

Listing 7.7 A basic _Layout.cshtml file calling `RenderBody`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewData["Title"]</title>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    @RenderBody()
</body>
</html>
```

ViewData is the standard mechanism for passing data to a layout from a view.

Elements common to every page, such as your CSS, are typically found in the layout.

Tells the templating engine where to insert the child view's content

As you can see, the layout file includes the required elements, such as `<html>` and `<head>`, as well as elements you need on every page, such as `<title>` and `<link>`. This example also shows the benefit of storing the page title in `ViewData`; the layout can render it in the `<title>` element so that it shows in the browser's tab, as shown in figure 7.11.

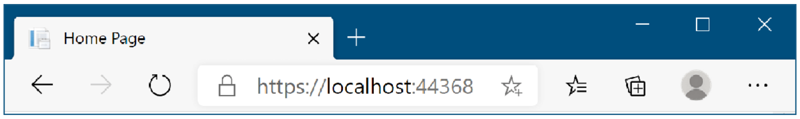


Figure 7.11 The contents of the `<title>` element is used to name the tab in the user's browser, in this case Home Page.

NOTE Layout files are not standalone Razor Pages and do not take part in routing, so they do not start with the `@page` directive.

Views can specify a layout file to use by setting the `Layout` property inside a Razor code block.

Listing 7.8 Setting the `Layout` property from a view

```
@{
    Layout = "_Layout";
```

Set the layout for the page to _Layout.cshtml.

```
    ViewData["Title"] = "Home Page";
}
<h1>@ViewData["Title"]</h1>
<p>This is the home page</p>
```

ViewData is a convenient way of passing data from a Razor view to the layout.

The content in the Razor view to render inside the layout

Any contents in the view will be rendered inside the layout, where the call to `@RenderBody()` occurs. Combining the two previous listings will result in the following HTML being generated and sent to the user.

Listing 7.9 Rendered output from combining a view with its layout

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Home Page</title>
    <link rel="stylesheet" href="/css/site.css" />
</head>
<body>
    <h1>Home Page</h1>
    <p>This is the home page</p>
</body>
<html>
```

ViewData set in the view is used to render the layout.

The RenderBody call renders the contents of the view.

Judicious use of layouts can be extremely useful in reducing the duplication between pages. By default, layouts only provide a single location where you can render content from the view, at the call to `@RenderBody`. In cases where this is too restrictive, you can render content using *sections*.

7.4.2 Overriding parent layouts using sections

A common requirement when you start using multiple layouts in your application is to be able to render content from child views in more than one place in your layout. Consider the case of a layout that uses two columns. The view needs a mechanism for saying "render *this* content in the *left* column" and "render this *other* content in the *right* column." This is achieved using *sections*.

> **NOTE** Remember, all of the features outlined in this chapter are specific to Razor, which is a server-side rendering engine. If you're using a client-side SPA framework to build your application, you'll likely handle these requirements in other ways, either within the client code or by making multiple requests to a Web API endpoint.

Sections provide a way of organizing where view elements should be placed within a layout. They're defined in the view using an `@section` definition, as shown in the following listing, which defines the HTML content for a sidebar separate from the main content, in a section called `Sidebar`. The `@section` can be placed anywhere in the file, top or bottom, wherever is convenient.

Listing 7.10 Defining a section in a view template

```
@{
    Layout = "_TwoColumn";
}
@section Sidebar {
    <p>This is the sidebar content</p>
}
<p>This is the main content </p>
```

All content inside the braces is part of the Sidebar section, not the main body content.

Any content not inside an @section will be rendered by the @RenderBody call.

The section is rendered in the parent layout with a call to `@RenderSection()`. This renders the content contained in the child section into the layout. Sections can be either required or optional. If they're required, a view *must* declare the given `@section`; if they're optional, they can be omitted, and the layout will skip them. Skipped sections won't appear in the rendered HTML. The following listing shows a layout that has a required section called `Sidebar` and an optional section called `Scripts`.

Listing 7.11 Rendering a section in a layout file, _TwoColumn.cshtml

```
@{
    Layout = "_Layout";
}
<div class="main-content">
    @RenderBody()
</div>
<div class="side-bar">
    @RenderSection("Sidebar", required: true)
</div>
@RenderSection("Scripts", required: false)
```

This layout is nested inside a layout itself.

Renders all the content from a view that isn't part of a section

Renders the Sidebar section; if the Sidebar section isn't defined in the view, throws an error

Renders the Scripts section; if the Scripts section isn't defined in the view, ignore it.

TIP It's common to have an optional section called `Scripts` in your layout pages. This can be used to render additional JavaScript that's required by some views, but that isn't needed on every view. A common example is the jQuery Unobtrusive Validation scripts for client-side validation. If a view requires the scripts, it adds the appropriate `@section Scripts` to the Razor markup.

You may notice that the previous listing defines a `Layout` property, even though it's a layout itself, not a view. This is perfectly acceptable and lets you create nested hierarchies of layouts, as shown in figure 7.12.

TIP Most websites these days need to be "responsive," so they work on a wide variety of devices. You generally *shouldn't* use layouts for this. Don't serve different layouts for a single page based on the device making the request. Instead, serve the same HTML to all devices, and use CSS on the client side to adapt the display of your web page as required.

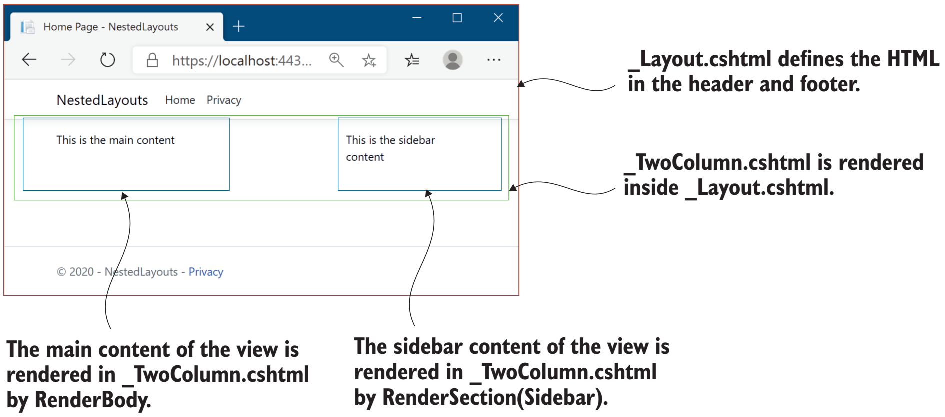

Home Page - NestedLayouts
https://localhost:443...
NestedLayouts Home Privacy
This is the main content
This is the sidebar content
© 2020 - NestedLayouts - Privacy
_Layout.cshtml defines the HTML in the header and footer.
_TwoColumn.cshtml is rendered inside _Layout.cshtml.
The main content of the view is rendered in _TwoColumn.cshtml by RenderBody.
The sidebar content of the view is rendered in _TwoColumn.cshtml by RenderSection(Sidebar).

Figure 7.12 Multiple layouts can be nested to create complex hierarchies. This allows you to keep the elements common to all views in your base layout and extract layout common to multiple views into sub-layouts.

Layout files and sections provide a lot of flexibility for building sophisticated UIs, but one of their most important uses is in reducing the duplication of code in your application. They're perfect for avoiding duplication of content that you'd need to write for every view. But what about those times when you find you want to reuse part of a view somewhere else? For those cases, you have partial views.

7.4.3 Using partial views to encapsulate markup

Partial views are exactly what they sound like—they're part of a view. They provide a means of breaking up a larger view into smaller, reusable chunks. This can be useful for both reducing the complexity in a large view by splitting it into multiple partial views, or for allowing you to reuse part of a view inside another.

Most web frameworks that use server-side rendering have this capability—Ruby on Rails has partial views, Django has inclusion tags, and Zend has partials. All of these work in the same way, extracting common code into small, reusable templates. Even client-side templating engines such as Mustache and Handlebars, used by client-side frameworks like Angular and Ember, have similar "partial view" concepts.

Consider a to-do list application again. You might find you have a Razor Page called ViewToDo.cshtml that displays a single to-do with a given `id`. Later on, you create a new Razor Page, RecentToDos.cshtml, that displays the five most recent to-do items. Instead of copying and pasting the code from one page to the other, you could create a partial view, called _ToDo.cshtml as in the following listing.

Listing 7.12 Partial view _ToDo.cshtml for displaying a `ToDoItemViewModel`

```
@model ToDoItemViewModel
<h2>@Model.Title</h2>
<ul>
    @foreach (var task in Model.Tasks)
    {
        <li>@task</li>
    }
</ul>
```

Partial views can bind to data in the Model property, like a normal Razor Page uses a PageModel.

The content of the partial view, which previously existed in the ViewToDo.cshtml file

Partial views are a bit like Razor Pages without the `PageModel` and handlers. Partial views are purely about rendering small sections of HTML, rather than handling requests, model binding, and validation, and calling the application model. They are great for encapsulating small usable bits of HTML that you need to generate on multiple Razor Pages.

Both the ViewToDo.cshtml and RecentToDos.cshtml Razor Pages can render the _ToDo.cshtml partial view, which handles generating the HTML for a single class. Partial views are rendered using the `<partial />` Tag Helper, providing the name of the partial view to render and the data (the model) to render. For example, the RecentToDos.cshtml view could achieve this as shown in the following listing.

Listing 7.13 Rendering a partial view from a Razor Page

This is a Razor Page, so it uses the @page directive. Partial views do not use @page.

```
@page
@model RecentToDoListModel

@foreach(var todo in Model.RecentItems)
{
    <partial name="_ToDo" model="todo" />
}
```

The PageModel contains the list of recent items to render.

Loop through the recent items. todo is a ToDoItemViewModel, as required by the partial view.

Use the partial tag helper to render the _ToDo partial view, passing in the model to render.

When you render a partial view without providing an absolute path or file extension, such as _ToDo in listing 7.13, the framework tries to locate the view by searching the Pages folder, starting from the Razor Page that invoked it. For example, if your Razor Page is located at Pages/Agenda/ToDos/RecentToDos.chstml, the framework would look in the following places for a file called _ToDo.chstml:

- Pages/Agenda/ToDos/ (the current Razor Page's folder)
- Pages/Agenda/
- Pages/
- Pages/Shared/
- Views/Shared/

The first location that contains a file called _ToDo.cshtml will be selected. If you include the .cshtml file extension when you reference the partial view, the framework will *only* look in the current Razor Page's folder. Also, if you provide an absolute path to the partial, such as /Pages/Agenda/ToDo.cshtml, that's the only place the framework will look.[4]

NOTE Like layouts, partial views are typically named with a leading underscore.

The Razor code contained in a partial view is almost identical to a standard view. The main difference is the fact that partial views are only called from other views. The other difference is that partial views don't run _ViewStart.cshtml when they execute, which you'll see shortly.

Child actions in ASP.NET Core

In the previous version of ASP.NET MVC, there was the concept of a *child action*. This was an action method that could be invoked from *inside* a view. This was the main mechanism for rendering discrete sections of a complex layout that had nothing to do with the main action method. For example, a child action method might render the shopping cart on an e-commerce site.

This approach meant you didn't have to pollute every page's view model with the view model items required to render the shopping cart, but it fundamentally broke the MVC design pattern by referencing controllers from a view.

In ASP.NET Core, child actions are no more. *View components* have replaced them. These are conceptually quite similar in that they allow both the execution of arbitrary code and the rendering of HTML, but they don't directly invoke controller actions. You can think of them as a more powerful partial view that you should use anywhere a partial view needs to contain significant code or business logic. You'll see how to build a small view component in chapter 20.

Partial views aren't the only way to reduce duplication in your view templates. Razor also allows you to pull common elements such as namespace declarations and layout configuration into centralized files. In the next section you'll see how to wield these files to clean up your templates.

7.4.4 Running code on every view with _ViewStart and _ViewImports

Due to the nature of views, you'll inevitably find yourself writing certain things repeatedly. If all of your views use the same layout, then adding the following code to the top of every page feels a little redundant:

```
@{
    Layout = "_Layout";
}
```

[4] As with most of Razor Pages, the search locations are conventions that you can customize if you wish. If you find the need, you can customize the paths as shown here: http://mng.bz/nM9e.

Similarly, if you find you need to reference objects from a different namespace in your Razor views, then having to add `@using WebApplication1.Models` to the top of every page can get to be a chore. Thankfully, ASP.NET Core includes two mechanisms for handling these common tasks: _ViewImports.cshtml and _ViewStart.cshtml.

IMPORTING COMMON DIRECTIVES WITH _VIEWIMPORTS

The _ViewImports.cshtml file contains directives that will be inserted at the top of every view. This includes things like the `@using` and `@model` statements that you've already seen—basically any Razor directive. To avoid adding a using statement to every view, you can include it in _ViewImports.cshtml instead of in your Razor Pages.

Listing 7.14 A typical _ViewImports.cshtml file importing additional namespaces

```
@using WebApplication1
@using WebApplication1.Pages
@using WebApplication1.Models
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The default namespace of your application and the Pages folder

Add this directive to avoid placing it in every view.

Makes Tag Helpers available in your views, added by default

The _ViewImports.cshtml file can be placed in any folder, and it will apply to all views and sub-folders in that folder. Typically, it's placed in the root Pages folder so that it applies to every Razor Page and partial view in your app.

It's important to note that you should *only* put Razor directives in _ViewImports .cshtml—you can't put any old C# in there. As you can see in the previous listing, this is limited to things like `@using` or the `@addTagHelper` directive that you'll learn about in the next chapter. If you want to run some arbitrary C# at the start of every view in your application, such as to set the `Layout` property, you should use the _ViewStart .cshtml file instead.

RUNNING CODE FOR EVERY VIEW WITH _VIEWSTART

You can easily run common code at the start of every Razor Page by adding a _View-Start.cshtml file to the Pages folder in your application. This file can contain any Razor code, but it's typically used to set the `Layout` for all the pages in your application, as shown in the following listing. You can then omit the `Layout` statement from all pages that use the default layout. If a view needs to use a non-default layout, you can override it by setting the value in the Razor Page itself.

Listing 7.15 A typical _ViewStart.cshtml file setting the default layout

```
@{
    Layout = "_Layout";
}
```

Any code in the _ViewStart.cshtml file runs before the view executes. Note that _View-Start.cshtml only runs for Razor Page views—it doesn't run for layouts or partial views.

Also note that the names for these special Razor files are enforced rather than conventions you can change.

> **WARNING** You must use the names _ViewStart.cshtml and _ViewImports.cshtml for the Razor engine to locate and execute them correctly. To apply them to all your app's pages, add them to the root of the Pages folder, not to the Shared subfolder.

You can specify additional _ViewStart.cshtml or _ViewImports.cshtml files to run for a subset of your views by including them in a subfolder in Pages. The files in the subfolders will run after the files in the root Pages folder.

Partial views, layouts, and AJAX

This chapter describes using Razor to render full HTML pages server-side, which are then sent to the user's browser in traditional web apps. A common alternative approach when building web apps is to use a JavaScript client-side framework to build a Single Page Application (SPA), which renders the HTML client-side in the browser.

One of the technologies SPAs typically use is AJAX (Asynchronous JavaScript and XML), in which the browser sends requests to your ASP.NET Core app without reloading a whole new page. It's also possible to use AJAX requests with apps that use server-side rendering. To do so, you'd use JavaScript to request an update for part of a page.

If you want to use AJAX with an app that uses Razor, you should consider making extensive use of partial views. You can then expose these via additional Razor Page handlers, as shown in this article: http://mng.bz/vzB1. Using AJAX can reduce the overall amount of data that needs to be sent back and forth between the browser and your app, and it can make your app feel smoother and more responsive, as it requires fewer full-page loads. But using AJAX with Razor can add complexity, especially for larger apps. If you foresee yourself making extensive use of AJAX to build a highly dynamic web app, you might want to consider using Web API controllers with a client-side framework (see chapter 9), or consider Blazor instead.

In this chapter I've focused on using Razor views with the Razor Page framework, as that's the approach I suggest if you're creating a server-side rendered ASP.NET Core application. However, as I described in chapter 4, you may want to use MVC controllers in some cases. In the final section of this chapter, we'll look at how you can render Razor views from your MVC controller actions, and how the framework locates the correct Razor view to render.

7.5 Selecting a view from an MVC controller

This section covers

- How MVC controllers use `ViewResults` to render Razor views
- How to create a new Razor view
- How the framework locates a Razor view to render

If you follow my advice from chapter 4, you should be using Razor Pages for your server-side rendered applications instead of the MVC controllers that were common in versions 1.x and 2.x. One of the big advantages Razor Pages gives is the close coupling of a Razor view to the associated page handlers, instead of having to navigate between multiple folders in your solution.

If for some reason you *do* need to use MVC controllers instead of Razor Pages, it's important to understand how you choose which view to render once an action method has executed. Figure 7.13 shows a zoomed-in view of this process, right after the action has invoked the application model and received some data back.

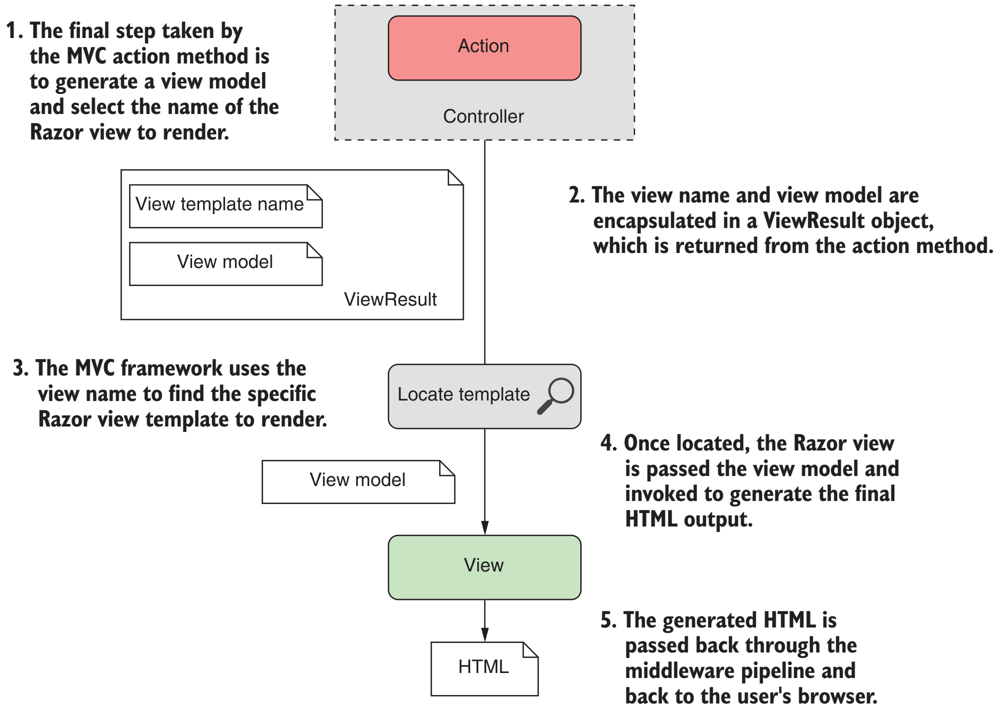

1. The final step taken by the MVC action method is to generate a view model and select the name of the Razor view to render.
Action
Controller
View template name
View model
ViewResult
2. The view name and view model are encapsulated in a ViewResult object, which is returned from the action method.
3. The MVC framework uses the view name to find the specific Razor view template to render.
Locate template
View model
4. Once located, the Razor view is passed the view model and invoked to generate the final HTML output.
View
HTML
5. The generated HTML is passed back through the middleware pipeline and back to the user's browser.

Figure 7.13 The process of generating HTML from an MVC controller using a `ViewResult`. This is very similar to the process for a Razor Page. The main difference is that for Razor Pages, the view is an integral part of the Razor Page; for MVC controllers, the view must be located at runtime.

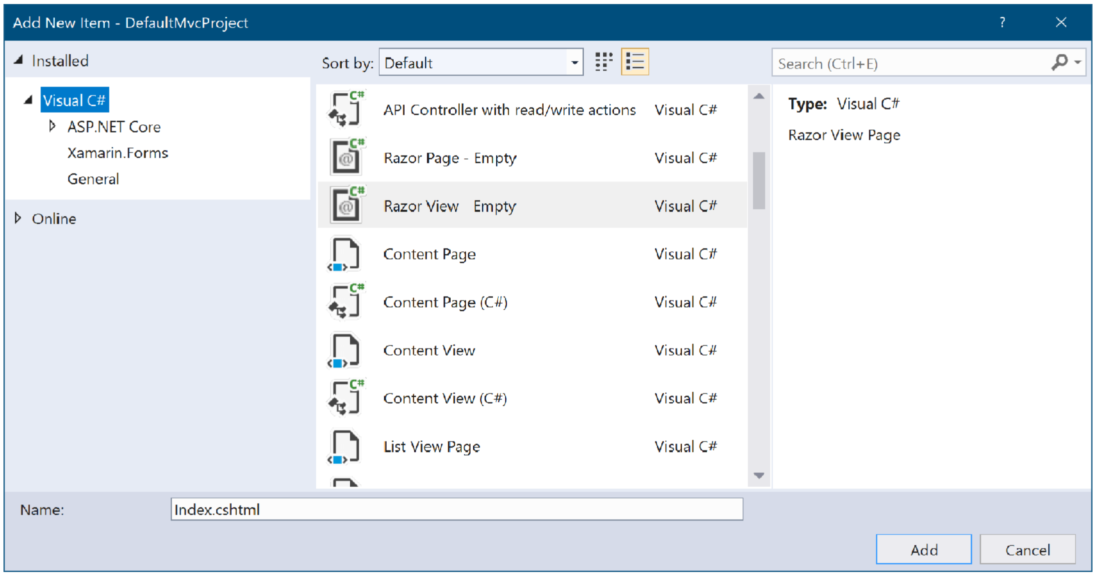
Some of this figure should be familiar—it's the lower half of figure 4.6 from chapter 4 (with a couple of additions) and is the MVC equivalent of figure 7.1. It shows that the MVC controller action method uses a `ViewResult` object to indicate that a Razor view should be rendered. This `ViewResult` contains the name of the Razor view template to render and a view model, an arbitrary POCO class containing the data to render.

NOTE I discussed `ViewResults` in chapter 4. They are the MVC equivalent of Razor Page's `PageResult`. The main difference is that a `ViewResult` includes a view name to render and a model to pass to the view template, while a `PageResult` always renders the Razor Page's associated view and passes the `PageModel` to the view template.

After returning a `ViewResult` from an action method, the control flow passes back to the MVC framework, which uses a series of heuristics to locate the view, based on the template name provided. Once a Razor view template has been located, the Razor engine passes the view model from the `ViewResult` to the view and executes the template to generate the final HTML. This final step, rendering the HTML, is essentially the same process as for Razor Pages.

You saw how to create controllers in chapter 4, and in this section you'll see how to create views and `ViewResult` objects and how to specify the template to render. You can add a new view template to your application in Visual Studio by right-clicking in an MVC application in Solution Explorer and choosing Add > New Item, and then selecting Razor View from the dialog, as shown in figure 7.14. If you aren't using Visual Studio, create a blank new file in the Views folder with the file extension .cshtml.

With your view template created, you now need to invoke it. In most cases you won't create a `ViewResult` directly in your action methods. Instead, you'll use one of the `View` helper methods on the `Controller` base class. These helper methods simplify passing in a view model and selecting a view template, but there's nothing magic about them—all they do is create `ViewResult` objects.


Add New Item - DefaultMvcProject
Installed
Visual C#
ASP.NET Core
Xamarin.Forms
General
Online
Search (Ctrl+E)
API Controller with read/write actions Visual C#
Razor Page - Empty Visual C#
Razor View Empty Visual C#
Content Page Visual C#
Content Page (C#) Visual C#
Content View Visual C#
Content View (C#) Visual C#
List View Page Visual C#
Add
Cancel

Sort by: Default

Type: Visual C#

Razor View Page

Name: Index.cshtml

Figure 7.14 The Add New Item dialog box. Choosing Razor View - Empty will add a new Razor view template file to your application.

In the simplest case you can call the `View` method without any arguments, as shown in the following listing. This helper method returns a `ViewResult` that will use conventions to find the view template to render, and will not supply a view model when executing the view.

Listing 7.16 Returning `ViewResult` from an action method using default conventions

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Inheriting from the Controller base class makes the View helper methods available.

The View helper method returns a ViewResult.

In this example, the `View` helper method returns a `ViewResult` without specifying the name of a template to run. Instead, the name of the template to use is based on the name of the controller and the name of the action method. Given that the controller is called `HomeController` and the method is called `Index`, by default the Razor template engine looks for a template at the Views/Home/Index.cshtml location, as shown in figure 7.15.

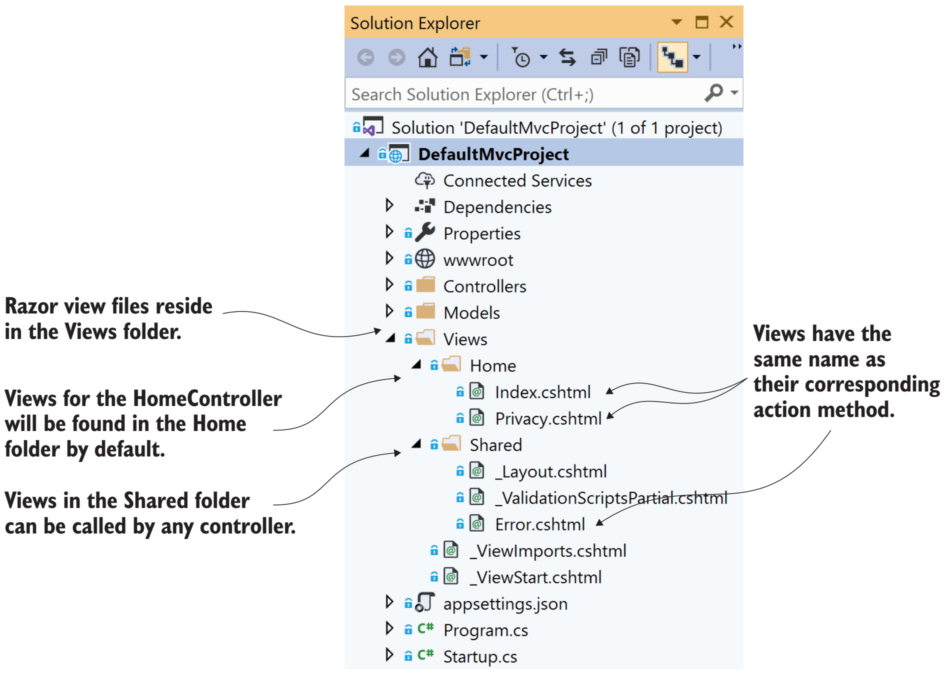

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'DefaultMvcProject' (1 of 1 project)
DefaultMvcProject
Connected Services
Dependencies
Properties
wwwroot
Controllers
Models
Views
Home
Index.cshtml
Privacy.cshtml
Shared
_Layout.cshtml
_ValidationScriptsPartial.cshtml
Error.cshtml
_ViewImports.cshtml
_ViewStart.cshtml
appsettings.json
Program.cs
Startup.cs
Razor view files reside in the Views folder.
Views for the HomeController will be found in the Home folder by default.
Views in the Shared folder can be called by any controller.
Views have the same name as their corresponding action method.

Figure 7.15 View files are located at runtime based on naming conventions. Razor view files reside in a folder based on the name of the associated MVC controller and are named with the name of the action method that requested them. Views in the Shared folder can be used by any controller.

This is another case of using conventions in MVC to reduce the amount of boilerplate you have to write. As always, the conventions are optional. You can also explicitly pass the name of the template to run as a `string` to the `View` method. For example, if the `Index` method instead returned `View("ListView")`, the templating engine would look for a template called ListView.cshtml instead. You can even specify the complete path to the view file, relative to your application's root folder, such as `View("Views/global.cshtml")`, which would look for the template at the Views/global.chtml location.

NOTE When specifying the absolute path to a view, you must include both the top-level Views folder and the .cshtml file extension in the path. This is similar to the rules for locating partial view templates.

The process of locating an MVC Razor view is very similar to the process of locating a partial view to render, as you saw in section 7.4. The framework searches in multiple locations to find the requested view. The difference is that for Razor Pages the search process only happens for *partial* view rendering, as the main Razor view to render is already known—it's the Razor Page's view template.

Figure 7.16 shows the complete process used by the MVC framework to locate the correct View template to execute when a `ViewResult` is returned from an MVC controller. It's possible for more than one template to be eligible, such as if an Index.chstml file exists in both the Home and Shared folders. Similar to the rules for locating partial views, the engine will use the first template it finds.

TIP You can modify all these conventions, including the algorithm shown in figure 7.16, during initial configuration. In fact, you can replace the whole Razor templating engine if required, but that's beyond the scope of this book.

You may find it tempting to explicitly provide the name of the view file you want to render in your controller; if so, I'd encourage you to fight that urge. You'll have a much simpler time if you embrace the conventions as they are and go with the flow. That extends to anyone else who looks at your code; if you stick to the standard conventions, there'll be a comforting familiarity when they look at your app. That can only be a good thing!

As well as providing a view template name, you can also pass an object to act as the view model for the Razor view. This object should match the type specified in the view's `@model` directive, and it's accessed in exactly the same way as for Razor Pages; using the `Model` property. The following listing shows two examples of passing a view model to a view.

Listing 7.17 Returning `ViewResult` from an action method using default conventions

```
public class ToDoController : Controller
{
    public IActionResult Index()
    {
        var listViewModel = new ToDoListModel();
        return View(listViewModel);
    }
```

Creating an instance of the view model to pass to the Razor view.

The view model is passed as an argument to View.

```
    public IActionResult View(int id)
    {
        var viewModel = new ViewToDoModel();
        return View("ViewToDo", viewModel);
    }

}
```

You can provide the view template name at the same time as the view model.

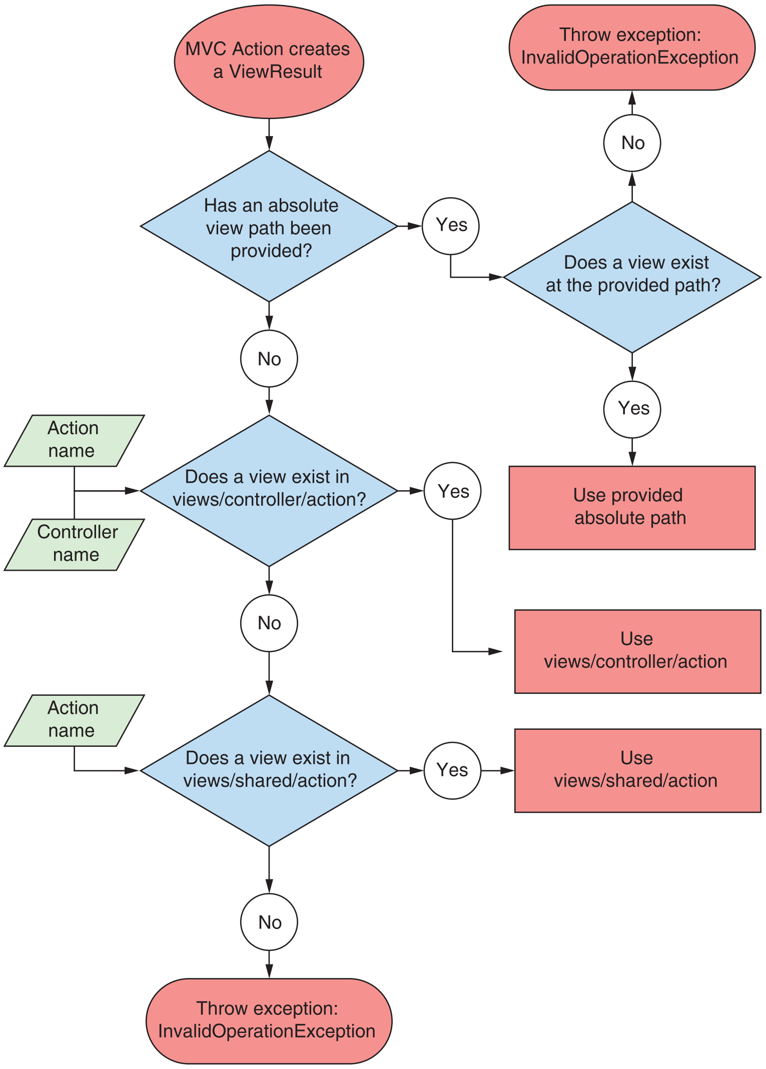

MVC Action creates a ViewResult
Has an absolute view path been provided?
Yes
No
Does a view exist at the provided path?
No
Throw exception: InvalidOperationException
Yes
Use provided absolute path
Action name
Controller name
Does a view exist in views/controller/action?
Yes
Use views/controller/action
No
Action name
Does a view exist in views/shared/action?
Yes
Use views/shared/action
No
Throw exception: InvalidOperationException

Figure 7.16 A flow chart describing how the Razor templating engine locates the correct view template to execute. Avoiding the complexity of this diagram is one of the reasons I recommend using Razor Pages wherever possible!

Once the Razor view template has been located, the view is rendered using the Razor syntax you've seen throughout this chapter. You can use all the features you've already seen—layouts, partial views, _ViewImports, and _ViewStart, for example. From the point of view of the Razor view, there's no difference between a Razor Pages view and an MVC Razor view.

That concludes our first look at rendering HTML using the Razor templating engine. In the next chapter you'll learn about Tag Helpers and how to use them to build HTML forms, a staple of modern web applications. Tag Helpers are one of the biggest improvements to Razor in ASP.NET Core over the previous version, so getting to grips with them will make editing your views an overall more pleasant experience!

Summary

- In the MVC design pattern, views are responsible for generating the UI for your application.
- Razor is a templating language that allows you to generate dynamic HTML using a mixture of HTML and C#.
- HTML forms are the standard approach for sending data from the browser to the server. You can use Tag Helpers to easily generate these forms.
- Razor Pages can pass strongly typed data to a Razor view by setting public properties on the `PageModel`. To access the properties on the view model, the view should declare the model type using the `@model` directive.
- Page handlers can pass key-value pairs to the view using the `ViewData` dictionary.
- Razor expressions render C# values to the HTML output using `@` or `@()`. You don't need to include a semicolon after the statement when using Razor expressions.
- Razor code blocks, defined using `@{}`, execute C# without outputting HTML. The C# in Razor code blocks must be complete statements, so it must include semicolons.
- Loops and conditionals can be used to easily generate dynamic HTML in templates, but it's a good idea to limit the number of `if` statements in particular, to keep your views easy to read.
- If you need to render a `string` as raw HTML you can use `Html.Raw`, but do so sparingly—rendering raw user input can create a security vulnerability in your application.
- Tag Helpers allow you to bind your data model to HTML elements, making it easier to generate dynamic HTML while staying editor friendly.
- You can place HTML common to multiple views in a layout. The layout will render any content from the child view at the location `@RenderBody` is called.
- Encapsulate commonly used snippets of Razor code in a partial view. A partial view can be rendered using the `<partial />` tag.
- _ViewImports.cshtml can be used to include common directives, such as `@using` statements, in every view.

- _ViewStart.cshtml is called before the execution of each Razor Page and can be used to execute code common to all Razor Pages, such as setting a default layout page. It doesn't execute for layouts or partial views.
- _ViewImports.cshtml and _ViewStart.cshtml are hierarchical—files in the root folder execute first, followed by files in controller-specific view folders.
- Controllers can invoke a Razor view by returning a `ViewResult`. This may contain the name of the view to render and optionally a view model object to use when rendering the view. If the view name is not provided, a view is chosen using conventions.
- By convention, MVC Razor views are named the same as the action method that invokes them. They reside either in a folder with the same name as the action method's controller or in the Shared folder.